Tiled Polymorphic Temporal Media

Paul Hudak, David Janin

LaBRI, CNRS UMR 5800
Laboratoire Bordelais de Recherche en Informatique

Rapport de recherche RR-1479-14

Tiled Polymorphic Temporal Media
– A Functional Pearl –

Mars 2014

Paul Hudak
Department of Computer Science
Yale University
New Haven, CT 06520-8285
paul.hudak@@yale.edu

David Janin
LaBRI
IPB, Université de Bordeaux
F-33405 Talence
david.janin@@labri.fr

Contents

Tiled Polymorphic Temporal Media
– A Functional Pearl –

Paul Hudak*
Department of Computer Science
Yale University
New Haven, CT 06520-8285
paul.hudak@yale.edu

David Janin †
LaBRI
IPB, Université de Bordeaux
F-33405 Talence
david.janin@labri.fr

March 3, 2014

Abstract

Tiled Polymorphic Temporal Media (Tiled PTM) is an algebraic approach to specifying the composition of multimedia values having an inherent temporal quality—for example sound clips, musical scores, computer animations, and video clips. Mathematically, one can think of a tiled PTM as a tiling in the one dimension of *time*. A tiled PTM value has two synchronization marks that specify, via an effective notion of *tiled product*, how the tiled PTMs are positioned in time relative to one another, possibly with overlaps.

Together with a pseudo inverse operation, and the related reset and coreset projection operators, the tiled product is shown to encompass both sequential and parallel products over temporal media. Up to observational equivalence, the resulting algebra of tiled PTM is shown to be an inverse monoid: the pseudo inverse being a semigroup inverse. These and other algebraic properties are explored in detail.

In addition, recursively-defined infinite tiles are considered. Ultimately, in order for a tiled PTM to be *renderable*, we must know its beginning, and how to compute its evolving value over time. Though undecidable in the general case, we define decidable special cases that still permit infinite tilings.

Finally, we describe an elegant specification, implementation, and proof of key properties in Haskell, whose lazy evaluation is crucial for assuring the soundness of recursive tiles. Illustrative examples, within the Euterpea framework for musical temporal media, are provided throughout.

*Partially supported by NSF grant CCF-1302327.

†Partially supported by CNRS fellowship grant at INS2I and the project INEDIT, ANR-12-CORD-009.

1 Introduction

It is natural to want to combine multimedia objects such as sound clips, video clips, musical phrases, and animation sequences, into larger objects. Such a composition process is also desirable for certain classes of discrete automata, such as robot motions. We say that such objects, or values, are *temporal*, since to render them properly requires "playing" them over some interval of time. Temporal values might even be *infinite*, resulting in an infinitely long rendering.

We would like a composition method that is sound, effective, hierarchical, and efficient. One general approach to this problem is captured in *polymorphic temporal media* (PTM) as described in [16, 14]. In its most simplified form, PTM can be summarized as:

1. A notion of a *neutral* value, i.e. a zero in the algebra (e.g. transparency or silence).

2. A set of *primitive* temporal values (e.g. video clips or musical notes).

3. A *binary sequential composition operator* (:+:) such that $p_1 :+: p_2$ represents the temporal rendering of p_1 followed by that of p_2.

4. A *binary parallel composition operator* (:=:) such that $p_1 :=: p_2$ represents the temporal renderings of p_1 and p_2 in parallel.

PTM has a rich set of algebraic properties, such as the associativity of (:+:) and (:=:), and the commutativity of (:=:). Indeed, it can be shown that there there exists a set of axioms that form a sound and complete axiomatization of PTM [16, 14].

Despite its simplicity, elegance, and practicality (it is the basis, for example, of the computer music libraries Haskore [17] and Euterpea [15]), there are some shortcomings. First of all, from a semantical point of view, the interpretation of $p_1 :=: p_2$ is not obvious—one might choose an interpretation in which p_1 and p_2 begin at the same time, or in which they are centered in time, or even aligned at their end-points. And one must decide whether the interpretation ends when the shortest of p_1 and p_2 ends, or the longest, and so on.

Secondly, from the point of view of convenience and aesthetics, it is often desirable to separate the "logical" start and end points of a temporal value from the "actual" start and end points. For example, a video clip might have a brief "fade in" scene that precedes the point in time that we logically think of as its starting point, or in the case of music there might be a "pick up" (also called an *anacrusis*) that precedes the measure that is the logical beginning of a phrase. This lack of expressiveness leads to a lack of modularity, in that changes to the "prefix" or "suffix" of a PTM value can affect the definitions of other PTM values.

We describe in this paper a new approach called *tiled PTM* that elegantly solves both of these problems. A tiled PTM naturally separates logical from actual start and stop times, and there is a single, unambigous composition operator that avoids the problems with (:=:). In addition, tiled PTMs enjoy a

rich algebraic structure, and lead to an elegant and efficient implementation in Haskell.

In the remainder of this paper we first formally define the notion of tiled PTM, along with its salient algebraic properties. We give an encoding of tiled PTM in terms of PTM, leading to an elegant Haskell implementation. We then extend tiled PTM with several useful operators, leading to a characterization of tiled PTM as an inverse semigroup. We also discuss the possibility of defining recursive, infinite tiles, along with its computational implications. We close with a comparison to related work.

Although the work described is polymorphic with respect to the media type, for pedagoical purposes many of our running examples are from the domain of music. And although these applications have a multimedia flavor, our methods are also applicable to other domains such a discrete automation. Our focus is on the methods that we use, and the formal algebraic properties, which we believe are more broadly applicable. All of the formal properties in the paper can be proved straightforwardly via equational reasoning in Haskell, although we omit the details.

2 Tiled PTM

Simply said, a tiled PTM is a temporal media value enriched with two synchronization marks *pre* and *post*, modeled as positive rationals representing their distances from the beginning of the temporal media. Such a tiled PTM is depicted in Figure 1. In a given tiled PTM, the *pre* synchronization mark specifies

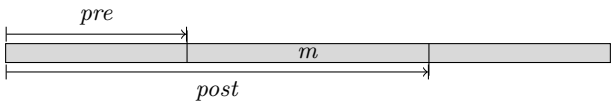

Figure 1: A tiled PTM

how a tiled PTM that comes *before* the given tile will be positioned. The *post* synchronization mark specifies how a tiled PTM that comes *after* the given tile will be positioned. The *synchronization* of tiled PTM is handled via a simple product called *tiled product*, which does two things: (1) it *synchronizes* the underlying PTMs in time, and (2) it *merges* (or *fuses*) the two PTM values positioned in such a way.

Remark. Of course, such a merge operation on temporal media values is dependent on the concrete temporal media type that is used. In the case of graphics, it might blend the two images; in the case of sound, it might mix the two signals; and so on.

More formally, synchronization is done by inserting some "neutral" values so that, when aligned, the $post_1$ synchronization mark on the first temporal media value matches the pre_2 synchronization mark on the second temporal media

value, which are then merged. The resulting *pre* synchronization mark of the product is set to the pre_1 synchronization mark of the first tile, and the resulting *post* synchronization mark of the product is set to the $post_2$ synchronization mark of the second tile. Such a tiled product is depicted in Figure 2.

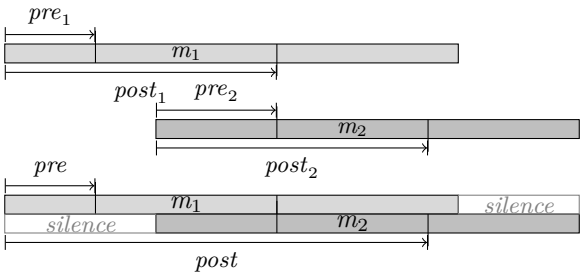

Figure 2: A tiled product instance

A key point about the tiled product is that *overlaps* may occur. The tiled product is thus neither a sequential product nor a parallel product—it is *both*. This dual feature of the tiled product is especially clear when observing, as done below, that the sequential and parallel products in PTM can be encoded, when lifted to tiled PTM, by means of the single tiled product.

Remark. By allowing *pre* and *post* synchronization marks to occur in arbitrary order, that is, also allowing *post* to occur before *pre* as we propose in the specification below, the tiled PTM type induces a rich algebraic structure, namely an *inverse semigroup*. Developed in abstract algebra in the middle of the 20th century, the theory of inverse semigroups [25] has been scarcely used in Computer Science. A notable exception is providing adequate notions for fine grain analysis of lambda calculus reduction [10, 1]. The notion of tiled PTM developed here provides evidence that this theory can be further applied even in concrete applications such as temporal media programming.

3 Specification and Implementation of Tiled PTM

We provide an illustrative implementation of tiled PTM that is based on *Euterpea* [15], a computer music library written in Haskell, where *Music* values play the role of temporal media. But it should be clear that other temporal media types could be used as well.

In music, *silence* is a temporal media value that acts as a neutral element when merged with other values, as mentioned in the previous section. Silence exists at all points in time falling before or after a *Music* value. In Euterpea, silence is represented as a rest: the value *rest d* is silence of duration *d*. When aiming at tiling other media types, one should seek for such a neutral value with respect to the fusion operation. For example, in the case of video it might be

a notion of *transparency*, and in discrete automation, a notion of a *idling* (the absence of a control event), and so on.

3.1 The Tile Data Type

The type *Tile* of tiled PTM is defined as follows.

$$\begin{aligned}&\textbf{data } \mathit{Tile}\ a = \mathit{Tile}\ \{\mathit{preT}\ \ :: \mathit{Dur},\\ &\qquad\qquad\qquad\qquad\ \ \mathit{postT} :: \mathit{Dur},\\ &\qquad\qquad\qquad\qquad\ \ \mathit{musT} :: \mathit{Music}\ a\}\end{aligned}$$

with **type** $\mathit{Dur} = \mathit{Rational}$.[1]

A function *durT* computes the *logical duration* of a tile; that is, the time delay between the synchronization mark *pre* and the synchronization mark *post*.

$$\begin{aligned}&\mathit{durT} :: \mathit{Tile}\ a \to \mathit{Dur}\\ &\mathit{durT}\ (\mathit{Tile}\ \mathit{pr}\ \mathit{po}\ m) = \mathit{po} - \mathit{pr}\end{aligned}$$

A tile t is a *positive tile* when $\mathit{durT}\ t$ is positive, it is a *negative tile* when $\mathit{durT}\ t$ is negative, and it is a *context tile* when it is neither positive nor negative, that is, when $\mathit{durT}\ t$ is zero.

Remark. Thanks to lazy evaluation, the temporal media value underlying a tiled PTM can be infinite. This feature is potentially preserved in tiled PTM, and means that temporal media values in tiles are implicitly completed by "silence" so that their duration is as least as large as both the distances defining the *pre* and *post* synchronization points.

3.2 Tiled Product

This is the key binary operator that combines two tiles by means of a synchronization followed by a fusion. For tiled PTM, it is defined as follows:

$$\begin{aligned}&(\%) :: \mathit{Tile}\ a \to \mathit{Tile}\ a \to \mathit{Tile}\ a\\ &\mathit{Tile}\ \mathit{pr}_1\ \mathit{po}_1\ m_1\ \%\ \mathit{Tile}\ \mathit{pr}_2\ \mathit{po}_2\ m_2 =\\ &\quad \textbf{let } d = \mathit{po}_1 - \mathit{pr}_2\\ &\quad \textbf{in } \mathit{Tile}\ (\mathit{max}\ \mathit{pr}_1\ (\mathit{pr}_1 - d))\ (\mathit{max}\ \mathit{po}_2\ (\mathit{po}_2 + d))\\ &\qquad\qquad (\textbf{if } d > 0 \textbf{ then } m_1 :=: \mathit{mDelay}\ d\ m_2\\ &\qquad\qquad\qquad\qquad \textbf{else } \mathit{mDelay}\ (-d)\ m_1 :=: m_2)\end{aligned}$$

where *mDelay* is defined by:

$$\begin{aligned}&\mathit{mDelay}\ d\ m = \textbf{case } \mathit{signum}\ d \textbf{ of}\\ &\quad 1 \quad \to \mathit{rest}\ d :+: m\\ &\quad 0 \quad \to m\\ &\quad -1 \to m :+: \mathit{rest}\ (-d)\end{aligned}$$

[1] We use Haskell's *Rational* numeric type to avoid the imprecision of, for example, floating point numbers, but mathematically the reals would be the most logical choice.

In other words, if d is positive, *mDelay* d m inserts d time units of silence in front of m, and if d is negative it inserts d units of silence after m. In the product of musical tiles, delays realize the synchronization of the music values, and the parallel composition realizes the fusion of the music values.

An example of a tiled product is already depicted in Figure 2, which illustrates the case $d > 0$ in the product definition. Another example is depicted in Figure 3, which illustrates the case $d < 0$ in the product definition. In this second

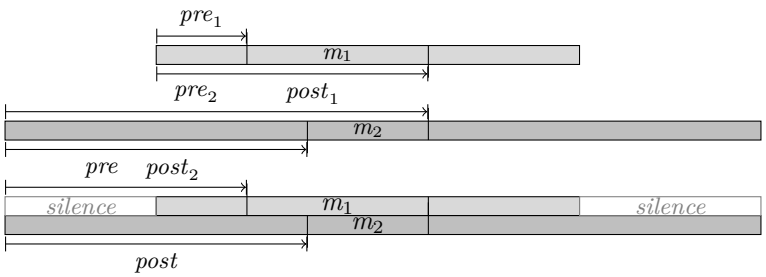

Figure 3: Another tiled product instance

example, we observe that the media value that will occur first in time comes from the second component of the product. Of course, such a phenomenon may occur in any configuration of tiles in a product, regardless of whether they are positive or negative.

3.3 Silent Tile

As mentioned earlier, *rest* is the neutral, or silent PTM value in Euterpea. We can lift this concept to tiled PTM by the defining:

$$
\begin{aligned}
&r :: \mathit{Dur} \rightarrow \mathit{Tile}\ a \\
&r\ d = \textbf{if}\ d < 0\ \textbf{then}\ \mathit{Tile}\ d\ 0\ (\mathit{rest}\ (-d)) \\
&\qquad \textbf{else}\ \mathit{Tile}\ 0\ d\ (\mathit{rest}\ d)
\end{aligned}
$$

Note that a silent tile may have a negative duration. The ramifications of this are discussed in a later section.

3.4 Observational Equivalence

In PTM, a notion of *observational equivalance* is defined, which is partially dependent on the underlying media type. We capture this equivalence here as the function *equiv*. Simply said, two PTMs are observationally equivalent when they have the same duration and are rendered the same way. The details about *equiv* can be found in [16, 14], but suffice it to say that it captures axioms such

as the following:

$$\begin{array}{rcl}
(m_1 :+: m_2) :+: m_3 & \text{`}equiv\text{`} & m_1 :+: (m_2 :+: m_3) \\
(m_1 :=: m_2) :=: m_3 & \text{`}equiv\text{`} & m_1 :=: (m_2 :=: m_3) \\
m_1 :=: m_2 & \text{`}equiv\text{`} & m_2 :=: m_1 \\
rest\ 0 :+: m & \text{`}equiv\text{`} & m \\
m :+: rest\ 0 & \text{`}equiv\text{`} & m
\end{array}$$

and, if $dur\ m_1 = dur\ m_3$, then

$$\begin{array}{l}
(m_1 :+: m_2) :=: (m_3 :+: m4\,) \\
\qquad\qquad \text{`}equiv\text{`}\ (m_1 :=: m_3) :+: (m_2 :=: m4\,),
\end{array}$$

In addition to these axioms, we assume that the following equivalence also holds:

$$m\ \text{`}equiv\text{`}\ (m :=: m)$$

for every PTM value m. Though a reasonable assumption, this requires abstracting from the intensity increase that, depending on the implementation, may arise when playing a melody twice at the same time.

We can lift this notion of observational equivalence to finite tiled PTM by defining:

$$\begin{array}{l}
Tile\ pr_1\ po_1\ m_1 \equiv Tile\ pr_2\ po_2\ m_2 = \\
\quad pr_1 - po_1 == pr_2 - po_2 \wedge \\
\quad \textbf{let}\ d\ \ = dur\ m_1 - dur\ m_2 \\
\qquad p\ \ = pr_1 - pr_2 \\
\qquad n_1 = \textbf{if}\ d < 0\ \textbf{then}\ mDelay\ d\ \ \ m_1\ \textbf{else}\ m_1 \\
\qquad n_2 = \textbf{if}\ d > 0\ \textbf{then}\ mDelay\ (-d)\ m_2\ \textbf{else}\ m_2 \\
\quad \textbf{in}\ \ \textbf{if}\ p > 0\ \textbf{then}\ n_1\ \text{`}equiv\text{`}\ mDelay\ p\ n_2 \\
\qquad\qquad\qquad \textbf{else}\ mDelay\ (-p)\ n_1\ \text{`}equiv\text{`}\ n_2
\end{array}$$

In other words, two finite tiled PTM are observationally equivalent when the distance between their synchronization marks are equal and their temporal media values are observationally equivalent when aligned on these synchronization marks, regardless of the duration of the silence that may occur before or after the temporal media.

With this notion of observational equivalence of tiled PTM, several properties become immediate. First of all, the tiled product (%) is associative; that is, for all tiled PTM values t_1, t_2, and t_3:

$$t_1 \,\%\, (t_2 \,\%\, t_3) \equiv (t_1 \,\%\, t_2) \,\%\, t_3$$

In addition, the 'silent' tiled PTM $(r\ 0)$ of duration 0 is a neutral element for the tiled product; that is, for all tiled PTM values t:

$$\begin{array}{l}
t \,\%\, r\ 0 \equiv t \\
r\ 0 \,\%\, t \equiv t
\end{array}$$

Therefore, up to observational equivalence, tiled PTMs equipped with tiled product form a *monoid*.

We also note that idempotent tiles coincide with context tiles. That is, for every tile t, we have:

$$t \,\%\, t \equiv t \quad \text{if and only if} \quad durT\ t = 0,$$

Moreover, these idempotent tiles commute. That is, for all tiled PTM t_1 and t_2, if both $durT\ t_1 = 0$ and $durT\ t_2 = 0$ then we have

$$t_1 \,\%\, t_2 \equiv t_2 \,\%\, t_1$$

3.5 Inverse, Reset and Co-Reset

Three functions that perform basic transformations on tile synchronization marks are the *inverse* (*inv*), the *reset* (*re*) and the *co-reset* (*co*) functions. They are defined as follows:

$$\begin{aligned}
&re, co, inv :: Tile\ a \to Tile\ a \\
&re\ \ (Tile\ pre\ post\ m) = Tile\ pre\ pre\ m \\
&co\ \ (Tile\ pre\ post\ m) = Tile\ post\ post\ m \\
&inv\ (Tile\ pre\ post\ m) = Tile\ post\ pre\ m
\end{aligned}$$

These functions are depicted in Figure 4. The functions *re* and *co*, restricted to

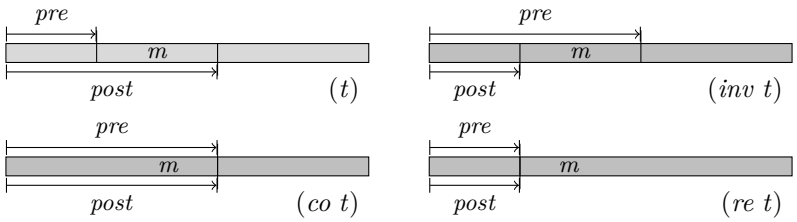

Figure 4: Reset, co-reset and inverse of a tile.

context tiles, are identities. It follows that these functions are projections from tiles to context tiles, and are idempotent; that is, for every tile t we have:

$$\begin{aligned}
re\ (re\ t) &= re\ t \\
co\ (co\ t) &= co\ t
\end{aligned}$$

As a consequence, *inv* acts as a duality operator since:

$$\begin{aligned}
re\ (inv\ t) &= co\ t \\
co\ (inv\ t) &= re\ t
\end{aligned}$$

The functions *re* and *co* could also be defined in terms of the function *inv* and the tiled product (%) since:

$$\begin{aligned} re\ t &\equiv t \,\%\, inv\ t \\ co\ t &\equiv inv\ t \,\%\, t \end{aligned}$$

In addition, we note that a tile that has been reset, but whose original duration is known, can be restored to its prior state. The same can be said for a tile that has been co-reset. More formally:

$$\begin{aligned} t &= re\ t \,\%\, r\ (durT\ t) \\ t &= r\ (durT\ t) \,\%\, co\ t \end{aligned}$$

(Recall that $(r\ d)$ is the 'silent' tile of duration d.)

3.6 Inverse Semigroup

In inverse semigroup theory (see [25]), the *semigroup inverse* of an element x is an element y such that:

$$\begin{aligned} x \cdot y \cdot x &= x, \text{ and} \\ y \cdot x \cdot y &= y \end{aligned}$$

In tiled PTM, the *inv* function does just that. Specifically, the following observational equivalences are satisfied:

$$\begin{aligned} t \,\%\, inv\ t \,\%\, t &\equiv t \\ inv\ t \,\%\, t \,\%\, inv\ y &\equiv inv\ t \end{aligned}$$

for all tiles t. The tiles depicted in Figure 4 are positioned in such a way that the above equations can easily be checked.

In general, an element may have several semigroup inverses. A semigroup with at least one inverse per element is called a *regular semigroup*. A semigroup where every element has exactly one inverse is called an *inverse semigroup*. As is well known in inverse semigroup theory [25], these semigroups exactly correspond to regular semigroups with commuting idempotence.

We have already seen in Section 3.4 that idempotent tiled PTMs commute. It follows that, up to observational equivalence, tiled PTMs form an inverse semigroup, and moreover form an *inverse monoid* since there is also a neutral element.

As in any inverse semigroup, the inverse operation is an anti-morphism. That is, for every tile t_1 and t_2 we have:

$$inv\ (t_1 \,\%\, t_2) \equiv inv\ t_2 \,\%\, inv\ t_1$$

However, this does not mean that the underlying PTMs are reversed as clearly shown by the product and inverse definitions.

3.7 Tiled product vs sequential or parallel product

We have announced above that the tiled product shares characteristics of both the sequential and the parallel product. This clearly appears in the figures above depicting the tiled product behavior. Such a statement can even be stated more precisely.

For the sequential product case, let $mToT$ be the function defined by

$$\begin{aligned} &mToT :: Music\ a \to Tile\ a \\ &mToT\ m = Tile\ 0\ (dur\ m)\ m \end{aligned}$$

that simply embeds any finite temporal media into a tiled temporal media with the same duration.

Then, for all finite PTMs m_1 and m_2, we have:

$$mToT\ (m_1 :+: m_2) \equiv mToT\ m_1\ \%\ mToT\ m_2$$

Since the function $mToT$ is clearly one-to-one up to behavioral equivalence, this property states that the monoid of finite PTMs equipped with the sequential product can be embedded into the monoid of tiled PTMs equipped with tiled product.

For the parallel product case, let $iMToT$ be the function defined by

$$\begin{aligned} &iMToT :: Music\ a \to Tile\ a \\ &iMToT\ m = Tile\ 0\ 0\ m \end{aligned}$$

that simply maps any finite or infinite temporal media into an idempotent tiled temporal media.

Then, for all finite or infinite PTM m_1 and m_2, we have

$$iMToT\ (m_1 :=: m_2) \equiv iMToT\ m_1\ \%\ iMToT\ m_2$$

This property states that there is a monoid homomorphism from the monoid of finite and infinite PTMs equipped with the parallel product and the monoid of tiled PTMs equipped with tiled product. It is worth noting that, up to observational equivalence and contrary to the mapping $mToT$, the mapping $iMTOT$ is not a one-to-one mapping.

Indeed, the two PTMs of the form m and m:+:$rest\ d$ for some strictly positive duration d are properly not observationally equivalent, while their images by the functions $iMToT$ are. Still, such a fact is already well known in inverse semigroup theory and is related with the notion of $\mathcal{R}$-equivalence in semigroup theory.[2]

More precisely, we observe that when m_1 and m_2 are both finite, then we have:

$$iMToT\ m_1 == re\ (mToT\ m_1) \text{ and } iMToT\ m_2 == re\ (mToT\ m_2)$$

[2]Two elements x and y are $\mathcal{R}$ equivalent in a semigroup S when there exists $z_1, z_2 \in S$ such that $x = yz_1$ and $y = xz_2$.

It follows that, as is well known in (inverse) semigroup theory, the PTMs m_1 and m_2 have equivalent images under the function $iMToT$ if and only if their images under $mToT$ are $\mathcal{R}$-equivalent in the inverse monoid of tiled PTM.

Remark. Additionally, let us mention that the above embeddings of finite and infinite PTM into tiled PTM have been studied and generalized quite in depth [12] in the setting of finite and infinite word language theory.

4 Further Embeddings in Euterpea

Since our tiled PTM is implemented in Euterpea, we can experiment with "musical tiles."

4.1 Primitive Tiles

Every *note function* in *Euterpea*, such as c, d, e, cs, ef, etc. (corresponding to the notes C, D, E, C♯ and E♭, respectively), can be turned into a tile. For example, instead of writing

$$c\ 4\ en :: Music\ Pitch$$

in Euterpea, which represents the note C in the fourth octave with duration of an eighth note, we can write

$$t\ c\ 4\ en :: Tile\ Pitch$$

using the following definition of the function t:

$$\begin{aligned} &t :: (Octave \to Dur \to Music\ Pitch) \\ &\quad \to Octave \to Dur \to Tile\ Pitch \\ &t\ n\ o\ d = \textbf{if}\ d < 0\ \textbf{then}\ Tile\ d\ 0\ (n\ o\ (-d)) \\ &\qquad\qquad \textbf{else}\ Tile\ 0\ d\ (n\ o\ d) \end{aligned}$$

4.2 Negative Durations

The definition of musical tiles with negative duration can be explained by the following equation that holds for every rational d:

$$t\ n\ o\ (-d) \equiv inv\ (t\ n\ o\ d)$$

In other words, negative durations allow one to invert the synchronization marks, but the music flow remains in the same direction.

The case of silent tiles is even a bit deeper since, for every rational d, d_1 and d_2:

$$\begin{aligned} r\ (-d) &\equiv inv\ (r\ d) \\ r\ d_1\ \%\ r\ d_2 &\equiv r\ (d_1 + d_2) \end{aligned}$$

In other words, the silent tiles equipped with the tiled product form a group isomorphic to the group $(\mathcal{Q}, +, 0)$ of rationals $\mathcal{Q}$ with sum.

4.3 Tempo

To change the tempo of a tile, we need to change the synchronization time and duration, as well as the tempo of the underlying *Music* value. This is performed by the function $tempoT$ defined by:

$$\begin{aligned}&tempoT :: Dur \to Tile\ a \to Tile\ a\\&tempoT\ r\ (Tile\ pr\ po\ m) =\\&\quad assert\ (r > 0)\ (Tile\ (pr/r)\ (po/r)\ (tempo\ r\ m))\end{aligned}$$

Here we use Haskell's *Control.Exception.assert* function to establish that the tempo scaling value r is positive.

4.4 Function Lift

Any Euterpea function on *Music* values can be lifted to a function on *Tile*s:

$$\begin{aligned}&liftT :: (Music\ a \to Music\ b) \to (Tile\ a \to Tile\ b)\\&liftT\ f\ (Tile\ pr\ po\ m) = Tile\ pr\ po\ (f\ m)\end{aligned}$$

Remark. $liftT\ (tempo\ r)$ is not quite the same as $tempoT\ r$!

4.5 From Tiles to Music

Of course, our goal in defining a tiled musical PTM is to eventually play it. Playing a tiled PTM amounts to converting the tile into a *Music* value, which can then be played in Euterpea. This transformation is captured by the following function.

$$\begin{aligned}&tToM :: Tile\ a \to Music\ a\\&tToM\ (Tile\ pr\ po\ m) = takeM\ (po - pr)\ (dropM\ pr\ m)\end{aligned}$$

where Euterpea's *takeM* and *dropM* on *Music* values are like *take* and *drop* on lists, except that they take duration arguments instead of list lengths.

Then, playing a tile is simply defined by the following composition.

$$playT = play \circ tToM$$

where *play* is Euterpea's function for playing a *Music* value.

Remark. In this definition, playing a tile amounts to extracting the part of the music value that goes *from* its *pre* synchronization mark *to* its *post* synchronization mark. This means in particular that negative tiles produce no sound at all. We note also that playing a tile will terminate even if, as it will be the case in Section 6, the underlying music value is infinite.

The parts of the underlying music value that lie outside of the *pre* and *post* synchronization marks can still be played by a suitable composition, via a tiled

product, with another tile, say simply rests. This allows one, for example, to define background music that is played until the lead music is over.

The function $tToM$ further allows us to relate the PTM semantics with the tiled PTM semantics as captured by the following observational equivalences. For all tiled PTM t_1 and t_2, we have

$$t_1 \equiv t_2$$
$$\text{if, and only if,}$$
$$tToM\ (c_1\ \%\ t_1\ \%\ c_2)\ `equiv`\ tToM\ (c_1\ \%\ t_2\ \%\ c_2)$$

for every other tiled PTM c_1 and c_2.

Remark. In the above statement, one can even restrict to silent tiles of the form $c_1 = r\ d_1$ and $c_2 = r\ d_2$ with durations $d_1 \geqslant 0$ and $d_2 \geqslant 0$.

4.6 An Example

We provide here a toy musical example, based on the jazz standard *There is no Greater Love*, by Isham Jones. Even if the reader is not familiar with the tune, certain advantages of the tiled PTM approach will be apparent, in particular the value of distinguishing between logical and actual start and stop times.

Indeed, in a tiled music piece, the *actual* start and stop times are modeled by the start and stop times of the music itself. The synchronization marks *pre* and *post* model instead the *logical* start and stop times of the music. Distinguishing these two notions in tiled PTM allows for a more abstract description of PTM structures, be they musical PTM or of any other kind.

In our example, the melody of the first eight bars of *There is no Greater Love* can be described in three *logical* phrases: the first four bars ($bar1'4$), the next two bars ($bar5'6$), and last two bars ($bar7'8$). But each of these phrases has a "pick up," or anacrusis, i.e. a short (in each case, three note) melody that precedes the start of the phrase. This distinction is captured below by aligning the pick-ups (pu_1, pu_2, and pu_3) with the logical phrases, using a co-reset:

$$\begin{aligned}
pu_1 &= t\ a\ 5\ en\ \%\ t\ bf\ 5\ en\ \%\ t\ c\ 6\ en \\
bar1'4 &= co\ pu_1 \\
&\quad \%\ t\ bf\ 5\ qn\ \%\ t\ a\ 5\ qn\ \%\ t\ g\ 5\ qn\ \%\ t\ d\ 5\ qn \\
&\quad \%\ t\ f\ 5\ qn\ \%\ t\ ff\ 5\ qn\ \%\ t\ ef\ 5\ qn\ \%\ t\ bf\ 4\ qn \\
&\quad \%\ t\ d\ 5\ (wn + qn)\ \%\ r\ dhn \\
pu_2 &= t\ d\ 5\ qn\ \%\ t\ a\ 5\ qn\ \%\ t\ af\ 5\ qn \\
bar5'6 &= co\ pu_2 \\
&\quad \%\ t\ g\ 5\ (wn + qn)\ \%\ r\ dhn \\
pu_3 &= t\ g\ 5\ qn\ \%\ t\ d\ 6\ qn\ \%\ t\ df\ 6\ qn \\
bar7'8 &= co\ pu_3 \\
&\quad \%\ t\ c\ 6\ wn\ \%\ r\ wn
\end{aligned}$$

These three phrases can then be combined to define the first eight bars of the tune:

$$tingl = bar1'4 \ \% \ bar5'6 \ \% \ bar7'8$$

A key point of this approach is that changes to the rhythm of the pick-ups can be made *without changing any other code*. For example, pu_2 can be changed to:

$$pu_2 = t \ d \ 5 \ en \ \% \ t \ a \ 5 \ en \ \% \ t \ af \ 5 \ en$$

which reduces the duration of the pick-up to three eight notes instead of three quarter notes. But because the pick-up is "outside" of the logical duration of $bar5'6$, no other changes are necessary. From a programming language perspective, this is simply an example of *modularity* in the design.

Of course, once a composition is complete, in order to render the result, one must ensure that any desired pick-ups at the very beginning (or suffixes at the very end) are heard. In the case above, this is easily accomplished as follows:

$$main = playT \ (r \ dhn \ \% \ tingl)$$

where *dhn* is a dotted half note (i.e. three quarter notes, the duration of pu_1).

This concludes our modeling example. A more detailed discussion of the relevance of tiled modeling for music is out of the scope of the present paper, but is adressed more fully in [19] and [23].

5 Other Useful Tile Functions

We provide here other useful functions that, either modifying synchronization marks while preserving the underlying music (like *resync* and *co-resync*), or modifying the underlying music while preserving the synchronization marks (like *stretch* and *co-stretch*), allow for experimenting with the structure of time in a tiled PTM.

5.1 Resync and Co-Resync

We consider here functions that move the synchronization marks with an invariant underlying temporal media. These functions come in a pair, the function *resync* preserving the relative position of the *pre* mark, the function *coresync* preserving the relative position of the *post* mark.

The functions *resync* and *coresynch* are defined as follows:

```
resync :: Dur → Tile a → Tile a
resync s (Tile pre post m) =
  let npost = post + s
  in if npost < 0
     then Tile (pre − npost) 0 (mDelay (−npost) m)
```

```
      else Tile pre npost m
coresync :: Dur → Tile a → Tile a
coresync s (Tile pre post m) =
  let npre = pre + s
  in if npre < 0
     then Tile 0 (post − npre) (mDelay (−npre) m)
     else Tile npre post m
```

These functions generalize the functions *re* and *co* in the sense that:

$$\begin{aligned} re\ (Tile\ pr\ po\ m) &= resync\ (pr - po)\ (Tile\ pr\ po\ m) \\ co\ (Tile\ pr\ po\ m) &= coresync\ (po - pr)\ (Tile\ pr\ po\ m) \end{aligned}$$

Their behaviors is depicted Figure 5 with $s > 0$.

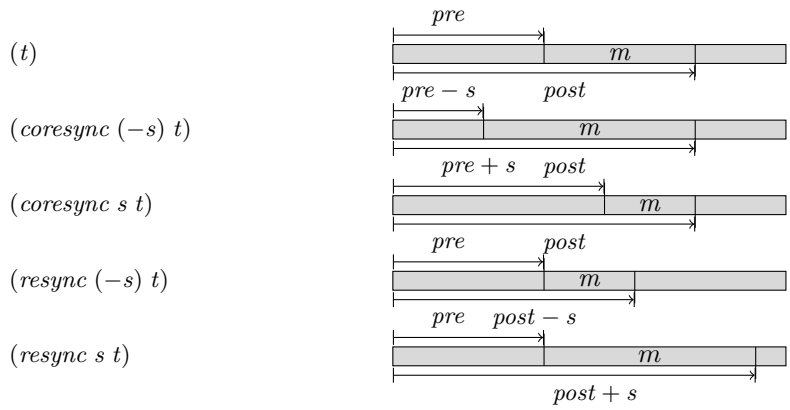

Figure 5: Resynchronization and co-resynchronization

Remark. The functions *resync* and *coresync* define *group actions* (in the algebraic sense) over the group $(\mathcal{Q}, +, 0)$ of rationals $\mathcal{Q}$ with sum. That is, for every tile t and every rational a and b, we have:

$$\begin{aligned} resync\ 0\ t &= t \\ resync\ a\ (resync\ b\ t) &= resync\ (a + b)\ t \\ coresync\ 0\ t &= t \\ coresync\ a\ (coresync\ b\ t) &= coresync\ (a + b)\ t \end{aligned}$$

As a consequence, we observe that, for every tile t and every duration offset s, we have:

$$\begin{aligned} resync\ s\ t &\equiv t\ \%\ r\ s \\ coresync\ s\ t &\equiv r\ s\ \%\ t \end{aligned}$$

5.2 Tile Resynchronization Examples

An example of resynchronization is *tile insertion* of two kinds. First, a *parallel fork* of a tile t_2 in a tile t_1 at a position d from its *pre* synchronization mark.

$$\begin{aligned}&\mathit{insertT} :: \mathit{Dur} \to \mathit{Tile}\ a \to \mathit{Tile}\ a \to \mathit{Tile}\ a\\&\mathit{insertT}\ d\ t_1\ t_2 = \mathit{coresync}\ (-d)\ (\mathit{re}\ t_2 \,\%\, \mathit{coresync}\ d\ t_1)\end{aligned}$$

Similarly, we can define tile *co-insertion* that is quite similar, but amounts to a *parallel join* instead.

$$\begin{aligned}&\mathit{coinsertT} :: \mathit{Dur} \to \mathit{Tile}\ a \to \mathit{Tile}\ a \to \mathit{Tile}\ a\\&\mathit{coinsertT}\ d\ t_1\ t_2 = \mathit{resync}\ (-d)\ (\mathit{resync}\ d\ t_1 \,\%\, \mathit{co}\ t_2)\end{aligned}$$

The behavior of these functions is depicted Figure 6. The function *insertT*

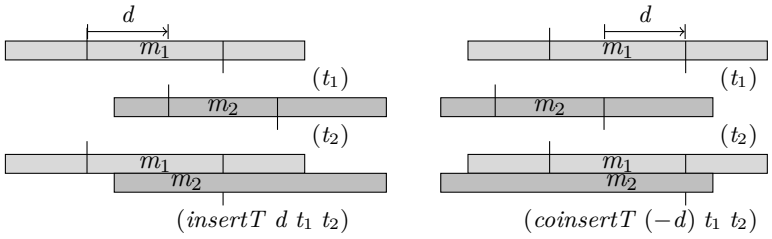

Figure 6: Fork and join tile insertions

behaves as a parallel fork in the sense that the inserted tile is synchronized on its *pre* synchronization mark. In contrast, the function *coinsertT* behaves as a parallel join in the sense that the inserted tile is synchronized on its *post* synchronization mark.

These functions illustrate especially well how context tiles, obtained by resets or co-resets of arbitrary tiles, can be used as adequately synchronized background media (say music) values.

Remark. Though we present here a direct encoding, we observe that these two functions can be derived from the tiled product % and the reset and co-reset functions in the sense that the following observational equivalences hold. For every tiled PTM t_1 and t_2, and for every duration offset d, we have:

$$\begin{aligned}\mathit{insertT}\ d\ t_1\ t_2 &\equiv r\ d \,\%\, \mathit{re}\ t_2 \,\%\, r\ (-d) \,\%\, t_1\\ \mathit{coinsertT}\ d\ t_1\ t_2 &\equiv t_1 \,\%\, r\ d \,\%\, \mathit{co}\ t_2 \,\%\, r\ (-d)\end{aligned}$$

5.3 Stretch and Co-Stretch

We consider now functions that stretch the underlying temporal media value while preserving the distance between the synchronization marks. These func-

tions also come in a pair, the function *stretch* preserving the relative position of the *pre* mark on the tiled PTM, and the function *costretch* preserving the relative position of the *post* mark.

The functions *stretch* and *costretch* are defined by:

$$
\begin{aligned}
&\mathit{stretch} :: \mathit{Dur} \to \mathit{Tile}\ a \to \mathit{Tile}\ a \\
&\mathit{stretch}\ r\ (\mathit{Tile}\ \mathit{pre}\ \mathit{post}\ m) = \\
&\quad \mathit{assert}\ (r > 0) \\
&\qquad (\mathit{Tile}\ (\mathit{pre} * r)\ (\mathit{pre} * (r-1) + \mathit{post}) \\
&\qquad\quad (\mathit{tempo}\ (1/r)\ m)) \\
&\mathit{costretch} :: \mathit{Dur} \to \mathit{Tile}\ a \to \mathit{Tile}\ a \\
&\mathit{costretch}\ r\ (\mathit{Tile}\ \mathit{pre}\ \mathit{post}\ m) = \\
&\quad \mathit{assert}\ (r > 0) \\
&\qquad (\mathit{Tile}\ (\mathit{post} * (r-1) + \mathit{pre})\ (\mathit{post} * r) \\
&\qquad\quad (\mathit{tempo}\ (1/r)\ m))
\end{aligned}
$$

Remark. As with *resync* and *coresync*, the functions *stretch* and *costretch* define group actions (in the algebraic sense), but now over the group $(\mathcal{Q}^+, *, 1)$ of positive rationals $\mathcal{Q}^+$ with product. That is, for every tile t and every positive rational a and b, we have:

$$
\begin{aligned}
\mathit{stretch}\ 1\ t &= t \\
\mathit{stretch}\ a\ (\mathit{stretch}\ b\ t) &= \mathit{stretch}\ (a * b)\ t \\
\mathit{costretch}\ 1\ t &= t \\
\mathit{costretch}\ a\ (\mathit{costretch}\ b\ t) &= \mathit{costretch}\ (a * b)\ t
\end{aligned}
$$

The behavior of the *stretch* and *costretch* functions is depicted Figure 7 with a ratio $r > 1$ and denoting by $m * r$ the *Music* value $(\mathit{tempo}\ (1/r)\ m)$ and by m/r the *Music* value $(\mathit{tempo}\ r\ m)$.

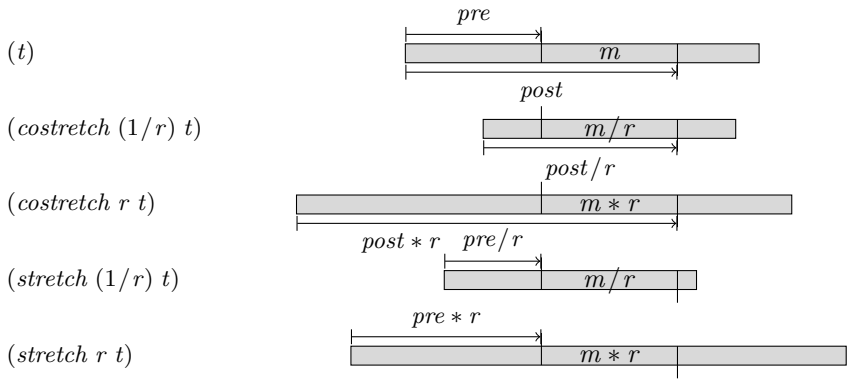

Figure 7: Stretch and co-stretch

5.4 Tile Stretching Examples

Since the *stretch* and *costretch* functions act on the tempo of the underlying PTM values while preserving the interval between synchronization marks, the example we propose comes from rhythm modeling. More precisely, the examples show how to generate basic waltz or salsa rhythms from a binary march rhythm.

$$
\begin{aligned}
&\mathit{march} = t\ c\ 4\ \mathit{qn} \,\%\, r\ \mathit{qn} \,\%\, t\ g\ 4\ \mathit{qn} \,\%\, r\ \mathit{qn}\\
&\mathit{waltz}\ \ = \mathit{costretch}\ (2/3)\ \mathit{march}\\
&\mathit{tumb} = \mathit{costretch}\ (5/4)\ \mathit{march}
\end{aligned}
$$

These three tiles are depicted in Figure 8 below. We observe that, in an implicit 3/4 metric, the *waltz* tile plays a note on the 2nd and 3rd beats; the two notes from the *march* tile have been "pushed" by 1/3 towards the *post* synchronization mark. In contrast, in an implicit 4/4 metrics, the *tumb* tile plays a note on the

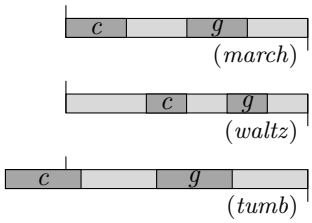

Figure 8: Rhythmical cells generated by stretching

4th beat of the preceding bar and on the 2nd after beat; the two notes from the *march* tile have been "pulled" by 1/4 before the *pre* synchronization mark. The resulting tile is a typical salsa bass riff: the *tumbao*.

The rhythms induced by these transformations can be heard when repeatedly playing these tiles in parallel with a basic percussion rhythm. For example:

$$
\begin{aligned}
&\mathit{bass}\ \ = \mathit{liftT}\ (\mathit{instrument}\ \mathit{Percussion})\\
&\qquad\qquad (\mathit{Tile}\ 0\ \mathit{wn}\ (\mathit{perc}\ \mathit{AcousticBassDrum}\ \mathit{wn}))\\
&\mathit{hiHat} = \mathit{liftT}\ (\mathit{instrument}\ \mathit{Percussion})\\
&\qquad\qquad (\mathit{Tile}\ 0\ (1/8)\ (\mathit{perc}\ \mathit{ClosedHiHat}\ (1/8)))\\
&\mathit{bassL}\ \ = \mathit{bass} \,\%\backslash\, \mathit{re}\ \mathit{bassL}\\
&\mathit{hiHatL} = \mathit{repeatT}\ 4\ \mathit{hiHat} \,\%\backslash\, \mathit{re}\ \mathit{hiHatL}\\
&\mathit{percL}\ \ = \mathit{re}\ \mathit{bassL} \,\%\, \mathit{hiHatL}\\
&\mathit{testW} = \mathit{playT}\ (\mathit{re}\ \mathit{bassL} \,\%\, \mathit{tempoT}\ (3/4)\ (\mathit{re}\ \mathit{hiHatL})\\
&\qquad\qquad\qquad\qquad \%\ \mathit{repeatT}\ 4\ \mathit{waltz})\\
&\mathit{testS}\ \ = \mathit{playT}\ (\mathit{re}\ \mathit{bassL} \,\%\, \mathit{re}\ \mathit{hiHatL} \,\%\, \mathit{repeatT}\ 4\ \mathit{tumb})
\end{aligned}
$$

Remark. In these examples, we make use of a special operator $\%\backslash$ that allows

defining infinite tiles as *bassL* and *hiHatL* above. These music tiles are used as unbounded background music. However, thanks to the definition of the player function *playT*, the execution of the resulting tiles is still finite. Various issues raised by the definition of infinite tiles are described in Section 6 below.

5.5 Another example

The canon song *frère Jacques* is defined first by describing its four basic verses.

$$
\begin{aligned}
&fj_1 = t\ c\ 4\ en \,\%\, t\ d\ 4\ en \,\%\, t\ e\ 4\ en \,\%\, t\ c\ 4\ en\\
&fj_2 = t\ e\ 4\ en \,\%\, t\ f\ 4\ en \,\%\, t\ g\ 4\ qn\\
&fj_3 = t\ g\ 4\ sn \,\%\, t\ a\ 4\ sn \,\%\, t\ g\ 4\ sn \,\%\, t\ f\ 4\ sn\\
&\quad \%\, t\ e\ 4\ en \,\%\, t\ c\ 4\ en\\
&fj_4 = t\ c\ 4\ en \,\%\, t\ g\ 3\ en \,\%\, t\ c\ 4\ qn
\end{aligned}
$$

As a first example, playing the first verse together with the second verse in parallel can be done as follows.

$$test_1 = playT\ (re\ fj_1 \,\%\, fj_2)$$

It is encoded by the "parallel fork" of the verse fj_1 with the verse fj_2. We observe that the same example can also be written as follows.

$$test_2 = playT\ (fj_1 \,\%\, co\ fj_2)$$

It is now encoded by the "join" of the verse fj_2 with the verse fj_1.

In these examples "fork" and "join" coincide over basic verses because the verses have the same length. Of course, this may not be true in general.

In *frère Jacques*, each verse is sung twice, so for convenience we define the function *repeatT* as follows.

$$
\begin{aligned}
&repeatT :: Integer \to Tile\ a \to Tile\ a\\
&repeatT\ n\ t = \mathbf{if}\ n \leqslant 0\ \mathbf{then}\ (r\ 0)\\
&\qquad\qquad\qquad\qquad \mathbf{else}\ t \,\%\, repeatT\ (n-1)\ t
\end{aligned}
$$

Then, the complete canon can be prepared as follows:

$$
\begin{aligned}
&fj = repeatT\ 2\ fj_1\\
&\quad \%\, re\ (repeatT\ 2\ fj_2 \,\%\, repeatT\ 2\ fj_3 \,\%\, repeatT\ 2\ fj_4)
\end{aligned}
$$

Worth noting is that the following example only plays twice the first verse:

$$test_3 = playT\ fj$$

Indeed, all other verses occur after the *post* synchronization mark of the tile *fj*.

Still, playing the entire melody can be done as follows.

$$test_4 = playT\ (fj\ \%\ r\ 6)$$

This just amounts to adding a rest of the duration of the six missing basic verses.

Last, the canon itself, that is, the basic melody that is launched in parallel over itself every two basic verses, can be performed as follows.

$$test_5 = playT\ (repeatT\ 4\ fj\ \%\ r\ 6)$$

Repeating the melody four times at least allows for hearing at least once all four distinct verses played in parallel. The rest added at the end of this last example allows for hearing the fade out inherent to the ending of every repetition.

6 Infinite Tiles

As briefly sketched in the exemples above, even though no rendering of a tiled PTM will actually take infinite time, there is still an interest in defining infinite tiled PTM, just as we might define infinite lists in other applications. In music, for instance, this means having the ability to handle unbounded background music that can be played in parallel with a finite tile. Our purpose in this section is to define these infinite tiles in the abstract and to see how the lazy evaluation mechanism of Haskell can be used to encode them.

6.1 Renderable infinite PTM

In Section 4.5, we refer to the *rendering* of a PTM (or tiled PTM) value m as the incremental elaboration of m's value over time, in such a way that the result can be presented to the user. That is, we want the result to be *played* (in the case of sound or music), *displayed* (in the case of video or animation), *executed* (in the case of discrete automation), and so on, depending on the underlying media type.

For something like $m_1 :+: m_2$ in conventional PTM, one can start rendering m_1 before knowing anything about m_2, since all of m_1 precedes m_2. This works even for infinite (in time) PTM values and allows for defining infinite PTM values by means of lazy evaluation. For example, the recursively defined, infinite PTM value:

$$m = c\ 4\ en :+: m$$

is well defined and renderable.

Even for something like $m_1 :=: m_2$, one can start rendering the result if one knows the first "renderable value" in m_1 and the first renderable value in m_2. So even if one or both of m_1 and m_2 is/are infinite, one can start rendering the result. Lazy evaluation in Haskell allows for defining renderable infinite PTMs in a simple and elegant way.

6.2 Renderable infinite tiled PTM

A *renderable infinite* tiled PTM is defined as renderable infinite PTM values equipped with two *finite* synchronization marks *pre* and *post*.

Then, following Section 4.5, observational equivalence is extended to infinite tiles as follows. We say that two finite or infinite tiles t_1 and t_2 are *observationally equivalent* when the two finite PTMs defined by

$$tToM\ (r\ d_1\ \%\ t_1\ \%\ tToM\ r\ d_2) \text{ and } tToM\ (r\ d_1\ \%\ t_2\ \%\ tToM\ r\ d_2)$$

are equivalent for all positive durations d_1 and d_2.

Despite such a simple definition, in the case of tiled PTMs, the direct recursive definition of a tiled PTM is not as simply done as for PTMs. For example, a recursive tiled PTM definition that is analogous to the one given above for ordinary PTM can be written:

$$x = t\ c\ 4\ en\ \%\ x$$

However, such an equation has *no solution* since the value of *post* will necessarily be infinite.

A subtler way to lift the above equation to tiled PTM would consists instead in defining the recursion on the *reset* of the variable tile as follows:

$$x = t\ c\ 4\ en\ \%\ re\ x$$

Such an equation clearly has a unique solution up to observational equivalence. However, its evaluation in Haskell does not terminate since it loops on the computation of the synchronization marks. This is because, in contrast to PTMs, in a tiled product such as $t_1 \,\%\, t_2$, the tile t_2 may have an "anacrusis" that starts *before* all of t_1 is rendered. This might even occur before the *pre* synchronization mark for t_1. But in order to determine that, we must know *all* of t_2, and therefore, if t_2 is infinite, this result is undecidable.

In other words, while infinite tiled PTMs can be defined via the "tiling" of infinite PTMs, we still lack of an effective and direct way to define infinite tiled PTMS from basic finite tiled PTMs. In the remainder of this section we analyze this problem in more detail and provide some effective solutions.

6.3 Recursive definition of tiled PTMs

In general we wish to provide a semantics and an implementation to tiled equations of the form:

$$x = f\ x$$

for some function $f :: Tile\ a \to Tile\ a$.

In general, such an equation may have no solution, or a unique solution, or even infinitely many solutions, and, clearly, such a problem is undecidable.

Indeed, the *pre* and *post* synchronization marks of the left occurrence of x may depend on the *pre* and *post* synchronization marks of the right occurrence

of x, hence creating an endless evaluation loop as illustrated by the equation $x = t\ c\ 4\ en\ \%\ re\ x$ above. So, it is likely that, as a Haskell program, the evaluation of the equation $x = f\ x$ will loop.

In the T-calculus approach [23] (see also [12] for an analysis of finite and infinite tiles in a language theoretical setting), the problem of solving an equation of the form $x = f\ x$ is handled by means of two successive phases.

We first provide sufficient conditions for the existence of a solution x with:

$$preT\ (x) = s \text{ and } postT\ (x) = s + d.$$

for some *computable* $s \geqslant 0$ and $d{>}0$. That is, we provide sufficient conditions for the computability of the *synchronization marks* of a strictly positive solution.

Then, a canonical solution (i.e. the least fixed point, if it exists) with such synchronization marks is defined in the usual way, by an iteration that computes the limit:

$$x = limf^n(r\ d)$$

The T-calculus is restricted enough in order to guarantee that such an iteration process is finite and converges. Every definable function f is in fact monotonic with respect to some adequate ordering over finite and infinite tiles: namely, the point-wise ordering over tiles obtained from defining the silent media value as the least possible media value, that is, the value $\perp$ in the domain-theoretic sense.

In the more general setting provided by our implementation in Haskell, such a solution is no longer available. The problem we aim at solving is thus split into two sub-problems.

The first one amounts to identifying a large class of functions over tiles such that, up to equivalence, the above equation has a unique abstract solution.

The second amounts to providing an Haskell encoding of this equation so that, in the adequate cases, its lazy evaluation computes a concrete renderable tiled PTM that represents this abstract solution.

We can do this as follows, first in the simplest case of equations solved by *iteration* of a single product, then in the more general case of fixpoint of so-called *very nice* functions.

6.4 Equations of the form $x = t\ \%\ re\ x$

The simplest recursive problems we wish to solve are equations of the form:

$$x = t\ \%\ re\ x$$

with a finite tile t. As already mentioned, such an equation has a unique solution in the case t is a positive tile. This means that we aim at defining a function $iterateT$ that takes any finite positive tile t as input and produces a renderable infinite tile $iterateT\ t$ such that the equivalence:

$$iterateT\ t \equiv t\ \%\ re\ (iterateT\ t)$$

is satisfied.

The production of a renderable infinite tile $iterateT\ t$ is in fact solvable by first defining the following alternate version of (%), called the *left restricted product*, or simply restricted product (%\):

$$
\begin{aligned}
&(\%\backslash) :: Tile\ a \to Tile\ a \to Tile\ a \\
&Tile\ pr_1\ po_1\ m_1\ \ \%\backslash\ \sim(Tile\ pr_2\ po_2\ m_2) = \\
&\quad Tile\ pr_1\ po_1 \\
&\qquad (m_1 :=: mDelay\ po_1\ (dropM\ pr_2\ m_2))
\end{aligned}
$$

Remark. In the restricted product (%\), the reset of the second argument is implicitly taken, and all the music that occurs before the synchronization marks is dropped.

It follows that, thanks to the "lazy pattern" (i.e. the use of tilde on the second argument), the computation of the *pre* and *post* synchronization marks of a product of the form $t_1 \,\%\backslash\, t_2$ only depends on t_1 and is done without evaluating the *pre* and *post* synchronization marks of the second component t_2.

This implies in particular that an equation of the form

$$x = y \,\%\backslash\, re\ x$$

with a positive tile y is now solvable over tiled PTMs by the lazy evaluation mechanisms in Haskell much in the same way an equation of the form $x = y :+: x$ is solvable over PTMs.

How such a product can be used to solve arbitrary equations of the form $x = y \,\%\, re\ x$ is described in the remainder of this section.

We show that the restricted product may replace the tiled product when used together with the $shiftT$ function defined by:

$$
\begin{aligned}
&shiftT :: Dur \to Tile\ a \to Tile\ a \\
&shiftT\ s\ t = resync\ s\ (coresync\ s\ t)
\end{aligned}
$$

Indeed, we can show that for all finite or infinite renderable tiles t_1 and t_2, for all duration offsets s, if $s \geqslant preT\ t_2$ and $durT\ (t_2) == 0$, that is, the tile t_2 is idempotent, then we have:

$$shiftT\ s\ (t_1 \,\%\, t_2) = shiftT\ s\ t_1 \,\%\backslash\, shiftT\ s\ t_2$$

In other words, when the above conditions are satisfied, the tiled product (%) can be replaced by the restricted product (%\).

Moreover, we observe that the function $shiftT$ satisfies the following additional properties. For every tile t, t_1 and t_2 and duration offset s, we have:

$$
\begin{aligned}
shiftT\ s\ (t_1 \,\%\, t_2) &= shiftT\ s\ t_1 \,\%\, shiftT\ s\ t_2 \\
shiftT\ s\ (re\ t) &= re\ (shiftT\ s\ t) \\
shiftT\ s\ (co\ t) &= co\ (shiftT\ s\ t)
\end{aligned}
$$

In other words, the function *shiftT s* is functorial with respect to the tile product, reset and co-reset. Note that this also implies that it commutes with respect to sync and resync.

The desired function *iterateT* can then be defined by:

$$
\begin{array}{l}
\mathit{iterateT} :: \mathit{Tile}\ a \to \mathit{Tile}\ a \\
\mathit{iterateT}\ y = \\
\quad \textbf{let}\ s = \mathit{preT}\ y \\
\qquad\quad x = \mathit{shiftT}\ s\ (\mathit{shiftT}\ (-s)\ y\ \%\backslash\ \mathit{shiftT}\ (-s)\ (\mathit{re}\ x)) \\
\quad \textbf{in}\ x
\end{array}
$$

The properties of the restricted product and the shift function given above ensure that if t is a positive finite tile then the tiled PTM *iterateT t* is renderable and, moreover, it is a solution of the semantic equation $x \equiv t\ \%\ re\ x$.

Such a solution can be tested with our music examples as follows:

$$
\begin{array}{l}
\mathit{recT}_1 = \mathit{iterateT}\ \mathit{tumb} \\
\mathit{testS}_1 = \mathit{playT}\ (\mathit{re}\ \mathit{percL}\ \%\ \mathit{recT}_1\ \%\ r\ 4)
\end{array}
$$

with *tumb* and *percL* defined in Section 5.4.

Remark. The fact the function *shiftT* commutes with tiled product, reset, co-reset and even resync and co-resync, allows for generalizing such an approach to arbitrary equations of the form $x = g\ (re\ x)$ *provided* that functions g are only defined from basic finite tiled PTMs combined with these functions. However, this would require a preprocessing treatment of tiled PTM programs in order to rewrite such equations into the desired form. In the next section, we provide instead a fixpoint operator that allows for solving even broader class of fixpoint equations.

6.5 Equations of the form $x = f\ x$

Out goal is to provide a simple way to solve equations of the more general form $x = f\ x$ for some larger class of functions f on tiles. That is, we look for a fixpoint operator

$$\mathit{fixT} :: (\mathit{Tile}\ a \to \mathit{Tile}\ a) \to \mathit{Tile}\ a$$

such that, for all functions $f :: \mathit{Tile}\ a \to \mathit{Tile}\ a$ in the adequate class, if *fixT f* terminates, then we have:

$$\mathit{fixT}\ f \equiv f\ (\mathit{fixT}\ f)$$

As already mentioned, the first problem we have to solve is that the evaluation of a recursive definition of the form:

$$x = f\ x$$

is very likely to loop when computing the synchronization marks of the expected solution. One way to cope with such a loop is to restrict it to a class of functions

for which these synchronization marks can be computed before running into such a loop.

Generalizing the approach proposed in [23] leads us to the following definition. A function

$$f :: \mathit{Tile}\ a \to \mathit{Tile}\ a$$

is *nice* when it admits a fixpoint t such that

$$\begin{aligned} \mathit{durT}\ t &> 0 \\ \mathit{preT}\ t &= \mathit{preT}\ (r\ 0) \\ \mathit{postT}\ t &= \mathit{postT}\ (r\ 0) \end{aligned}$$

Remark. Restricting to strictly positive solutions will disallow a fixpoint equation such as $x = \mathit{re}\ x$ for which every idempotent tiled PTM is a solution.

Then the computation of such a fixpoint can be attempted as follows. We first define the function *forceSync* that forces the synchronization marks of a tile.

$$\begin{aligned} &\mathit{forceSync} :: \mathit{Dur} \to \mathit{Dur} \to \mathit{Tile}\ a \to \mathit{Tile}\ a \\ &\mathit{forceSync}\ \mathit{npr}\ \mathit{npo}{\sim}(\mathit{Tile}\ \mathit{pr}\ \mathit{po}\ m) = \\ &\quad \textbf{if} \quad \mathit{npr} < \mathit{npo} \\ &\quad \textbf{then}\ (\mathit{inv}\ (\mathit{forceSync}\ \mathit{npo}\ \mathit{npr}\ (\mathit{Tile}\ \mathit{po}\ \mathit{pr}\ m))) \\ &\quad \textbf{else}\ \ \mathit{Tile}\ \mathit{npr}\ \mathit{npo} \\ &\qquad\qquad (\textbf{if}\ \mathit{npr} < \mathit{pr}\ \textbf{then}\ \mathit{dropM}\ (\mathit{pr} - \mathit{npr})\ m \\ &\qquad\qquad\ \ \textbf{else}\ \mathit{rest}\ (\mathit{npr} - \mathit{pr}) :+: m) \end{aligned}$$

Then, assuming the function f is nice, computing a fixpoint amounts to solving the equation:

$$x = f\ (\mathit{forceSync}\ (\mathit{preT}\ (f\ (r\ 0)))\ (\mathit{postT}\ (f\ (r\ 0)))\ x)$$

provided the solution x that is computed that way is strictly positive with synchronization marks that are compatible with such an approximation in the sense that:

$$\mathit{preT}\ (f\ (r\ 0)) - \mathit{preT}\ x \geqslant 0 \text{ and } \mathit{durT}\ (f\ (r\ 0)) == \mathit{durT}\ x.$$

The first condition ensures that only silent values are dropped when applying the function *forceSync*. Together with the second condition that ensures the durations are equal, this guarantees that we have

$$x \equiv \mathit{forceSync}\ (\mathit{preT}\ (f\ (r\ 0)))\ (\mathit{postT}\ (f\ (r\ 0)))\ x$$

therefore x is indeed a solution of the equation $x \equiv f\ x$.

A fixpoint operator that performs such a computation is then simply defined as follows.

$$
\begin{aligned}
&\mathit{fixT} :: (\mathit{Tile}\ a \to \mathit{Tile}\ a) \to \mathit{Tile}\ a \\
&\mathit{fixT}\ f = \mathbf{let}\ \mathit{pr} = \mathit{preT}\ (f\ (r\ 0)) \\
&\qquad\qquad\ \ \mathit{po} = \mathit{postT}\ (f\ (r\ 0)) \\
&\qquad\qquad\ \ y\ \ = f\ (\mathit{forceSync}\ \mathit{pr}\ \mathit{po}\ y) \\
&\qquad\quad \mathbf{in}\ \mathit{assert}\ (\mathit{durT}\ y > 0 \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{pr} - \mathit{preT}\ y \geqslant 0 \\
&\qquad\qquad\qquad\qquad \wedge\ \mathit{po} - \mathit{pr} == \mathit{durT}\ y) \\
&\qquad\quad\ y
\end{aligned}
$$

The arguments developed above ensure that, for every nice function f over tiles, if *fixT* f terminates, raising no assertion violation, then we indeed have:

$$\mathit{fixT}\ f \equiv f\ (\mathit{fixT}\ f)$$

Remark. In the definition of *forceSync*, the use of the lazy pattern ensures an "evaluation upon need" of the *pre* and *post* synchronization marks of the tile argument. This prevents *forceSync* in the definition of *fixT* from entering an endless loop while computing the *pre* and *post* synchronization marks of y in the recursive definition $y = f\ (\mathit{forceSync}\ \mathit{pr}\ \mathit{po}\ y)$ in *fixT*. In other words, whenever it terminates, the tile *fixT* f is a renderable tile.

The solution proposed here can be tested with our music examples as follows:

$$
\begin{aligned}
&\mathit{tileReProd} :: \mathit{Tile}\ a \to \mathit{Tile}\ a \to \mathit{Tile}\ a \\
&\mathit{tileReProd}\ t_1\ t_2 = t_1 \,\%\, \mathit{re}\ t_2 \\
&\mathit{recT}_2 = \mathit{fixT}\ (\mathit{tileReProd}\ \mathit{tumb}) \\
&\mathit{testS}_2 = \mathit{playT}\ (\mathit{re}\ \mathit{percL} \,\%\, \mathit{recT}_2 \,\%\, r\ 4)
\end{aligned}
$$

This illustrates the fact that the solution proposed here generalizes the solution proposed in the previous section.

Remark. We refer to the class of nice functions that admit a fixpoint that can be computed as above as *very nice*. Of course, being very nice is clearly undecidable. Nevertheless, all simple examples we can think of that have a fixpoint with strictly positive duration are very nice. More precisely, we make the following conjectures:

Let *Simple* be the class of unary functions over tiles that can be built from constant functions $x \mapsto t$ and the function reset $x \mapsto \mathit{re}\ x$ with all basic operators and functions defined in the previous sections, that is, the (point wise extension of) the tiled product, reset, co-reset and inverse operators, as well as all stretching and resynchronization functions with constant rational parameters. Then every function $f \in \mathit{Simple}$ such that $\mathit{preT}(f(r0)) < \mathit{posT}(f(r0))$ is very nice.

7 Related Work

There has been considerable work on embedding semantic descriptions *in* multimedia (XML, UML, the Semantic Web, etc.), but not on formalizing the semantics *of* concrete media. There are also many authoring tools and scripting languages for designing multimedia applications. The one closest to a programming language is probably SMIL [5]. There are also dozens of computer music languages that have been proposed over the years, including [27, 26, 8, 29, 7, 2, 9, 13, 6, 30]. None of these languages, however, have a notion of tiling.

The idea of tiling temporal media rendering by means of *pre* and *post* synchronization marks first appeared in the abstract several decades ago in the language LOCO [11]. We have borrowed from this proposal the name of the synchronization marks. However, no systematic study of the induced algebra seems to have been done since then.

The notion of tiled temporal media itself has slowly been formalized in a serie of papers, starting from the modeling of rhythmic features in music representation [19], through an algebraic approach to audio or music pattern synchronization [3], an experimental audio implementation [24], and then a toy abstract calculus proposal: the T-calculus [23]. However, in such proposals, no real programmatic features such as control flow are available: the T-calculus essentially has the expressive power of finite sequential synchronous transducers [28].

Together with the notion of polymorphic temporal media [16, 14], the notion of tiled temporal media has reached maturity as described in this paper. Besides the conceptual elegance of PTM that can simply be lifted into tiled PTM, the concrete implementation of tiled PTM in Haskell/Euterpea [15] that is proposed in this paper allows for experimenting and applying the underlying concepts on a much broader scale.

8 Conclusions

In this paper, we have shown that embedding PTMs into tiled PTMs, which essentially amounts to *internalizing* synchronization features into PTMs, can be done in a simple and elegant way. The underlying algebraic semantics inherits many properties from the algebraic theory of inverse semigroups [25]. In particular, the sequential and parallel products that are needed in the temporal media are then merged into the single tiled product.

As illustrated by the anacrusis example, such an embedding allows for defining complex combinations of PTMs with a more abstract and modular point of view over the concrete synchronization mechanisms that such complex combinations may involve.

The capacity that is offered to define and handle renderable infinite tiled PTMs by means of the *fixT* operator provides an effective way for defining unbounded temporal media. The semantics of rendering infinite tiles by automatically extracting finite portions avoids the somewhat fragile necessity that might consist, in some other approaches, of explicitly killing infinite tiles when

no longer needed.

For future work, we observe that our tiled product places no constraints on its arguments. For example, two tiles that are incoherent, say with "harmonically incompatible" musical content, can still be combined. Though this makes no difference at a programatic level, such a lack of compatibility constraints could be a weakness in application with creative end-users. But it is known that the model of tiles can be extended with compatibility constraints, either in a linear time setting [21] or in a branching time setting [20]. This suggests that tiled PTM could be *annotated* or *tagged* by abstract description of their PTM values. For instance, *Music* values can be annotated by harmonic tags such as chord symbols. Then, compatibility constrains could be encoded in the tiled product of such enriched tiled PTM via tag-matching constraints.

Extending accordingly our implementation in Haskell would even allow us to handle higher-order tags with compatibility contraints solved by unification. This could lead to the development of some notion of higher-order tiled PTM with tiled product acting not only as a synchronization product but, also as a communication product. The availability of an underlying robust language theory [18, 22, 20, 4] could also allow handling higher-order tiled PTMs whose semantics would be define by means of (manageable) sets of concrete tiled PTMs

References

[1] S. Abramsky. A structural approach to reversible computation. *Theoretical Comp. Science*, 347(3):441–464, 2005.

[2] D.P. Anderson and R. Kuivila. Formula: A programming language for expressive computer music. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[3] F. Berthaut, D. Janin, and B. Martin. Advanced synchronization of audio or symbolic musical patterns: an algebraic approach. *International Journal of Semantic Computing*, 6(4):409–427, 2012.

[4] A. Blumensath and D. Janin. A syntactic congruence for languages of birooted trees. Technical Report RR-1478-14, LaBRI, Université de Bordeaux, 2014.

[5] Dick C.A. Bulterman and Lloyd Rutledge. *SMIL 3.0 – Interactive Multimedia for the Web, Mobile Devices and Daisy Talking Books*. X.media.publishing, 2008.

[6] P. Cointe and X. Rodet. Formes: an object and time oriented system for music composition and synthesis. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programmming*, pages 85–95. ACM, 1984.

[7] D. Collinge. Moxie: A languge for computer music performance. In *Proc. Int'l Computer Music Conference*, pages 217–220. Computer Music Association, 1984.

[8] R.B. Dannenberg. The Canon score language. *Computer Music Journal*, 13(1):47–56, 1989.

[9] R.B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[10] V. Danos and L. Regnier. Reversible, irreversible and optimal lambda-machines. *Theoretical Comp. Science*, 227(1-2):79–97, 1999.

[11] P. Desain and H. Honing. LOCO: a composition microworld in Logo. *Computer Music Journal*, 12(3):30–42, 1988.

[12] A. Dicky and D. Janin. Embedding finite and infinite words into overlapping tiles. Technical Report RR-1475-13, LaBRI, Université de Bordeaux, 2013.

[13] G. Haus and A. Sametti. Scoresynth: A system for the synthesis of music scores based on petri nets and a music algebra. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[14] P. Hudak. A sound and complete axiomatization of polymorphic temporal media. Technical Report RR-1259, Department of Computer Science, Yale University, 2008.

[15] P. Hudak. *The Haskell School of Music : From signals to Symphonies*. Yale University, Department of Computer Science, 2013.

[16] Paul Hudak. An algebraic theory of polymorphic temporal media. In *Proceedings of PADL'04: 6th International Workshop on Practical Aspects of Declarative Languages*, pages 1–15. Springer Verlag LNCS 3057, June 2004.

[17] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[18] D. Janin. Quasi-recognizable vs MSO definable languages of one-dimensional overlapping tiles. In *Mathematical Found. of Comp. Science (MFCS)*, volume 7464 of *LNCS*, pages 516–528, Bratislava, Slovakia, 2012.

[19] D. Janin. Vers une modélisation combinatoire des structures rythmiques simples de la musique. *Revue Francophone d'Informatique Musicale (RFIM)*, 2, 2012.

[20] D. Janin. Algebras, automata and logic for languages of labeled birooted trees. In *Int. Col. on Aut., Lang. and Programming (ICALP)*, volume 7966 of *LNCS*, pages 318–329, Riga, Latvia, 2013. Springer.

[21] D. Janin. On languages of one-dimensional overlapping tiles. In *Int. Conf. on Current Thrends in Theo. and Prac. of Comp. Science (SOFSEM)*, volume 7741 of *LNCS*, pages 244–256, Spindlerûv Mlýn, Czech Republic, 2013. Springer.

[22] D. Janin. Overlaping tile automata. In *8th International Computer Science Symposium in Russia (CSR)*, volume 7913 of *LNCS*, pages 431–443, Ekaterinburg, Russia, 2013. Springer.

[23] D. Janin, F. Berthaut, M. DeSainte-Catherine, Y. Orlarey, and S. Salvati. The T-calculus : towards a structured programming of (musical) time and space. In *ACM Workshop on Functional Art, Music, Modeling and Design (FARM)*, Boston, USA, 2013. ACM Press.

[24] D. Janin, F. Berthaut, and M. DeSainteCatherine. Multi-scale design of interactive music systems : the libTuiles experiment. In *10th Conference on Sound and Music Computing (SMC)*, 2013.

[25] M. V. Lawson. *Inverse Semigroups : The theory of partial symmetries*. World Scientific, 1998.

[26] O. Orlarey, D. Fober, S. Letz, and M. Bilton. Lambda calculus and music calculi. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1994.

[27] Yann Orlarey, Dominique Fober, and Stéphane Letz. Faust: an efficient functional approach to DSP programming. In *New Computationals Paradigms for Computer Music*. Editions Delatour France, 2009.

[28] J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[29] B. Schottstaedt. Pla: A composer's idea of a language. *Computer Music Journal*, 7(1):11–20, 1983.

[30] G. Wang, R. Fiebrink, and P. Cook. Combining analysis and synthesis in the ChucK programming language. In *Proceedings of the International Computer Music Conference*, 2007.